IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re the Application of:		TI-35424
	Serge Lasserre, et al.	Art Unit: 2186
Serial No:	10/632,076	Examiner:
Filed:	July 31, 2003	Conf. No.: 2218
For:	Memory Management of Local Variables Upon a Change of Context	

TRANSMITTAL LETTER ACCOMPANYING CERTIFIED COPY OF PRIORITY APPLICATION UNDER 35 U.S.C. §119

MAILING CERTIFICATE UNDER 37 C.F.R. §1.8(a)
I hereby certify that the above correspondence is being deposited with the U.S. Postal Service as First Class Mail in an envelope addressed to: Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450, on **March 12, 2004.**

Robin E. Barnum

Robin E. Barnum

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Dear Sir:

Submitted herewith is a certified copy of **European** Patent Application No. **03291907.8 (TI-35424EP)**, filed on **July 30, 2003**, in the **European** Patent Office and from which priority under 35 U.S.C. §119 is claimed for the above-identified application.

Respectfully submitted,

Robert D. Marshall, Jr.

Robert D. Marshall, Jr.
Attorney for Applicant
Reg. No. 28,527

Texas Instruments Incorporated
P.O. Box 655474, MS 3999
Dallas, TX 75265
(972) 917-5290

TI-35424EP
FOR: S/N 10/632,076
(TI-35424)

Europäisches Patentamt | **European Patent Office** | **Office européen des brevets**

Bescheinigung | Certificate | Attestation

Die angehefteten Unterlagen stimmen mit der ursprünglich eingereichten Fassung der auf dem nächsten Blatt bezeichneten europäischen Patentanmeldung überein.

The attached documents are exact copies of the European patent application described on the following page, as originally filed.

Les documents fixés à cette attestation sont conformes à la version initialement déposée de la demande de brevet européen spécifiée à la page suivante.

Patentanmeldung Nr. | Patent application No. | Demande de brevet n°

03291907.8

Der Präsident des Europäischen Patentamts;
Im Auftrag

For the President of the European Patent Office

Le Président de l'Office européen des brevets
p.o.

R C van Dijk

Anmeldung Nr:
Application no.: 03291907.8
Demande no:

Anmeldetag:
Date of filing: 30.07.03
Date de dépôt:

Anmelder/Applicant(s)/Demandeur(s):

Texas Instruments Inc.
7839 Churchill Way,
M/S 3999
Dallas,
Texas 75251
ETATS-UNIS D'AMERIQUE
Texas Instruments France
Avenue Bel Air BP5
06271 Villeneuve Loubet Cedex,
Nice
FRANCE

Bezeichnung der Erfindung/Title of the invention/Titre de l'invention:
(Falls die Bezeichnung der Erfindung nicht angegeben ist, siehe Beschreibung.
If no title is shown please refer to the description.
Si aucun titre n'est indiqué se referer à la description.)

Memory management of local variables upon a change of context

In Anspruch genommene Prioriät(en) / Priority(ies) claimed /Priorité(s) revendiquée(s)
Staat/Tag/Aktenzeichen/State/Date/File no./Pays/Date/Numéro de dépôt:

US/31.07.02/US 400391 P

Internationale Patentklassifikation/International Patent Classification/
Classification internationale des brevets:

G06F9/00

Am Anmeldetag benannte Vertragstaaten/Contracting states designated at date of filing/Etats contractants désignées lors du dépôt:

AT BE BG CH CY CZ DE DK EE ES FI FR GB GR HU IE IT LU MC NL
PT RO SE SI SK TR LI

MEMORY MANAGEMENT OF LOCAL VARIABLES UPON A CHANGE OF CONTEXT

BACKGROUND OF THE INVENTION

The present subject matter relates generally to processors and more particularly to managing local variables used by a processor. More particularly, the subject matter disclosed herein relates to reducing the number of external memory accesses upon changing from one thread to another.

Many types of electronic devices are battery operated and thus preferably consume as little power as possible. An example is a cellular telephone. Further, it may be desirable to implement various types of multimedia functionality in an electronic device such as a cell phone. Examples of multimedia functionality may include, without limitation, games, audio decoders, digital cameras, etc. It is thus desirable to implement such functionality in an electronic device in a way that, all else being equal, is fast, consumes as little power as possible and requires as little memory as possible. Improvements in this area are desirable.

BRIEF SUMMARY OF THE INVENTION

In some embodiments, a cache subsystem may comprise a multi-way set associative cache and a data memory that holds a contiguous block of memory defined by an address stored in a register. Local variables (e.g., Java local variables) may be stored in the data memory. The data memory preferably is adapted to store two groups of local variables. A first group comprises local variables associated with finished methods and a second group comprises local variables associated with unfinished methods.

Further, local values are saved to, or fetched from, external memory upon a context change based on a first value associated with the first and second groups. The first value may comprise a threshold address or an allocation bit associated with each of a plurality of lines forming the data memory.

NOTATION AND NOMENCLATURE

Certain terms are used throughout the following description and claims to refer to particular system components. As one skilled in the art will appreciate, semiconductor companies may refer to a component by different names. This document does not intend to distinguish between components that differ in name but not function. In the following discussion and in the claims, the terms "including" and "comprising" are used in an open-ended fashion, and thus should be interpreted to mean "including, but not limited to…". Also, the term "couple" or "couples" is intended to mean either an indirect or direct connection. Thus, if a first device couples to a second device, that connection may be through a direct connection, or through an indirect connection via other devices and connections.

BRIEF DESCRIPTION OF THE DRAWINGS

For a more detailed description of the preferred embodiments of the present invention, reference will now be made to the accompanying drawings, wherein:

Figure 1 shows a diagram of a system in accordance with preferred embodiments of the invention and including a Java Stack Machine ("JSM") and a Main Processor Unit ("MPU");

Figure 2 depicts an exemplary embodiment of the system described herein in the form of a communication device (e.g., cellular telephone);

Figure 3 shows a block diagram of the JSM of Figure 1 in accordance with a preferred embodiment of the invention;

Figure 4 shows various registers used in the JSM of Figures 1 and 3;

Figure 5 illustrates the storage of local variables and pointers in accordance with the preferred embodiments;

Figure 6 illustrates the use of the local variable pointers upon returning from a method; and

Figure 7 illustrates a preferred embodiment of cache-based data storage in the JSM of Figure 3;

Figure 8 illustrates the mapping of the cache's data array to main memory; and

Figure 9 illustrates the use of a pointer to boundary separating local variables associated with finished and unfinished methods; and

Figures 10 and 11 illustrate the use of an allocation bit to delineate the boundary between the local variables associated with finished and unfinished methods.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

The following discussion is directed to various embodiments of the invention. Although one or more of these embodiments may be preferred, the embodiments disclosed should not be interpreted, or otherwise used, as limiting the scope of the disclosure, including the claims, unless otherwise specified. In addition, one skilled in the art will understand that the following description has broad application, and the discussion of any embodiment is meant only to be exemplary of that embodiment, and

not intended to intimate that the scope of the disclosure, including the claims, is limited to that embodiment.

The subject matter disclosed herein is directed to a programmable electronic device such as a processor having memory in which "local variables" associated with a stack-based language (e.g., Java) and pointers associated with the local variables may be stored. The term "local variables" refers to temporary variables used by a method that executes on the processor. Multiple methods may run on the processor and each method preferably has its own set of local variables. In general, local variables have meaning only while their associated method is running. The stack-based language may comprise Java Bytecodes although this disclosure is not so limited. In Java Bytecodes, the notion of local variables ("LVs") is equivalent to automatic variables in other programming languages (e.g., "C") and other termed variables in still other programming languages.

The following describes the operation of a preferred embodiment of such a processor in which the methods and local variables may run and be used. Other processor architectures and embodiments may be used and thus this disclosure and the claims which follow are not limited to any particular type of processor. Details regarding the storage of the local variables and the associated pointers follow the description of the processor.

The processor described herein is particularly suited for executing Java™ Bytecodes, or comparable code. As is well known, Java is particularly suited for embedded applications. Java is a relatively "dense" language meaning that on average each instruction may perform a large number of functions compared to various other programming languages. The dense nature of Java is of particular benefit for portable, battery-operated devices that preferably include as little memory as possible to save space and

power. The reason, however, for executing Java code is not material to this disclosure or the claims that follow.

Referring now to Figure 1, a system 100 is shown in accordance with a preferred embodiment of the invention. As shown, the system includes at least two processors 102 and 104. Processor 102 is referred to for purposes of this disclosure as a Java Stack Machine ("JSM") and processor 104 may be referred to as a Main Processor Unit ("MPU"). System 100 may also include memory 106 coupled to both the JSM 102 and MPU 104 and thus accessible by both processors. At least a portion of the memory 106 may be shared by both processors meaning that both processors may access the same shared memory locations. Further, if desired, a portion of the memory 106 may be designated as private to one processor or the other. System 100 also includes a Java Virtual Machine ("JVM") 108, compiler 110, and a display 114. The JSM 102 preferably includes an interface to one or more input/output ("I/O") devices such as a keypad to permit a user to control various aspects of the system 100. In addition, data streams may be received from the I/O space into the JSM 102 to be processed by the JSM 102. Other components (not specifically shown) may be included as desired. As such, while system 100 may be representative of, or adapted to, a wide variety of electronic systems, an exemplary electronic system may comprise a battery-operated, mobile cell phone such as that shown in Figure 2. As shown in Figure 2, a mobile communications device includes an integrated keypad 412 and display 414. The JSM 102 and MPU 104 noted above and other components may be included in electronics package 410 which may be coupled to keypad 410, display 414, and radio frequency ("RF") circuitry 416 which may be connected to an antenna 418.

Referring again to Figure 1, as is generally well known, Java code comprises a plurality of "Bytecodes" 112. Bytecodes 112 may be provided to the JVM 108, compiled by compiler 110 and provided to the JSM 102 and/or MPU 104 for execution therein. In accordance with a preferred embodiment of the invention, the JSM 102 may execute at least some, and generally most, of the Java Bytecodes. When appropriate, however, the JSM 102 may request the MPU 104 to execute one or more Java Bytecodes not executed or executable by the JSM 102. In addition to executing Java Bytecodes, the MPU 104 also may execute non-Java instructions. The MPU 104 also hosts an operating system ("O/S") (not specifically shown), which performs various functions including system memory management, the system task management that schedules the JVM 108 and most or all other native tasks running on the system, management of the display 114, receiving input from input devices, etc. Without limitation, Java code may be used to perform any one of a variety of applications including multimedia, games or web based applications in the system 100, while non-Java code, which may comprise the O/S and other native applications, may still run on the system on the MPU 104.

The JVM 108 generally comprises a combination of software and hardware. The software may include the compiler 110 and the hardware may include the JSM 102. The JVM may include a class loader, bytecode verifier, garbage collector, and a bytecode interpreter loop to interpret the bytecodes that are not executed on the JSM processor 102.

In accordance with preferred embodiments of the invention, the JSM 102 may execute at least two instruction sets. One instruction set may comprise standard Java bytecodes. As is well-known, Java is a stack-based programming language in which instructions generally target a stack. For example, an integer

add ("IADD") Java instruction pops two integers off the top of the stack, adds them together, and pushes the sum back on the stack. As will be explained in more detail below, the JSM 102 comprises a stack-based architecture with various features that accelerate the execution of stack-based Java code.

Another instruction set executed by the JSM 102 may include instructions other than standard Java instructions. In accordance with at least some embodiments of the invention, such other instruction set may include register-based and memory-based operations to be performed. This other instruction set generally complements the Java instruction set and, accordingly, may be referred to as a complementary instruction set architecture ("C-ISA"). By complementary, it is meant that the execution of various "complex" Java Bytecodes may be substituted by "micro-sequences" using C-ISA instructions that permit address calculation to readily "walk through" the JVM data structures. Such micro-sequences also may use Bytecode instructions in addition to C-ISA instructions. The execution of Java may be made more efficient and run faster by replacing some sequences of Bytecodes by preferably shorter and more efficient sequences of C-ISA instructions. The two sets of instructions may be used in a complementary fashion to obtain satisfactory code density and efficiency. As such, the JSM 102 generally comprises a stack-based architecture for efficient and accelerated execution of Java bytecodes combined with a register-based architecture for executing register and memory based C-ISA instructions. Both architectures preferably are tightly combined and integrated through the C-ISA.

Figure 3 shows an exemplary block diagram of the JSM 102. As shown, the JSM includes a core 120 coupled to data storage 122 and instruction storage 130. The core may include one or more

components as shown. Such components preferably include a plurality of registers 140, three address generation units ("AGUs") 142, 147, micro-translation lookaside buffers (micro-TLBs) 144, 156, a multi-entry micro-stack 146, an arithmetic logic unit ("ALU") 148, a multiplier 150, decode logic 152, and instruction fetch logic 154. In general, operands may be retrieved from data storage 122 or from the micro-stack 146, processed by the ALU 148, while instructions may be fetched from instruction storage 130 by fetch logic 154 and decoded by decode logic 152. The address generation unit 142 may be used to calculate addresses based, at least in part on data contained in the registers 140. The AGUs 142 may calculate addresses for C-ISA instructions as will be described below. The AGUs 142 may support parallel data accesses for C-ISA instructions that perform array or other types of processing. The AGU 147 couples to the micro-stack 146 and may manage overflow and underflow conditions in the micro-stack preferably in parallel. The micro-TLBs 144, 156 generally perform the function of a cache for the address translation and memory protection information bits that are preferably under the control of the operating system running on the MPU 104.

Referring now to Figure 4, the registers 140 may include 16 registers designated as R0-R15. Registers R0-R3, R5, R8-R11 and R13-R14 may be used as general purposes ("GP") registers usable for any purpose by the programmer. Other registers, and some of the GP registers, may be used for specific functions. For example, registers R4 and R12 may be used to store two program counters. Register R4 preferably is used to store the program counter ("PC") and register R12 preferably is used to store a micro-program counter ("micro-PC"). In addition to use as a GP register, register R5 may be used to store the base address of a

portion of memory in which Java local variables may be stored when used by the currently executing Java method. The top of the micro-stack 146 may be referenced by the values in registers R6 and R7. The top of the micro-stack has a matching address in external memory pointed to by register R6. The values contained in the micro-stack are the latest updated values, while their corresponding values in external memory may or may not be up to date. Register R7 provides the data value stored at the top of the micro-stack. Registers R8 and R9 may also be used to hold the address index 0 ("AI0") and address index 1 ("AI1"), as will be explained below. Register R14 may also be used to hold the indirect register index ("IRI"). Register R15 may be used for status and control of the JSM 102.

Referring again to Figure 3, as noted above, the JSM 102 is adapted to process and execute instructions from at least two instruction sets and one or more methods comprising such instructions. One instruction set includes stack-based operations and the second instruction set includes register-based and memory-based operations. The stack-based instruction set may include Java Bytecodes. Java Bytecodes pop, unless empty, data from and push data onto the micro-stack 146. The micro-stack 146 preferably comprises the top *n* entries of a larger stack that is implemented in data storage 122. Although the value of *n* may be vary in different embodiments, in accordance with at least some embodiments, the size *n* of the micro-stack may be the top eight entries in the larger, memory-based stack. The micro-stack 146 preferably comprises a plurality of gates in the core 120 of the JSM 102. By implementing the micro-stack 146 in gates (e.g., registers) in the core 120 of the processor 102, access to the data contained in the micro-stack 146 is generally very fast, although any particular access speed is not a limitation on this

disclosure. The second register-based, memory-based instruction set may comprise the C-ISA instruction set introduced above. The C-ISA instruction set preferably is complementary to the Java bytecode instruction set in that the C-ISA instructions may be used to accelerate or otherwise enhance the execution of Java Bytecodes.

The ALU 148 adds, subtracts, and shifts data. The multiplier 150 may be used to multiply two values together in one or more cycles. The instruction fetch logic 154 generally fetches instructions from instruction storage 130. The instructions may be decoded by decode logic 152. Because the JSM 102 is adapted to process instructions from at least two instruction sets, the decode logic 152 generally comprises at least two modes of operation, one mode for each instruction set. As such, the decode logic unit 152 may include a Java mode in which Java instructions may be decoded and a C-ISA mode in which C-ISA instructions may be decoded.

The data storage 122 generally comprises data cache ("D-cache") 124 and a data random access memory ("D-RAMset") 126. Reference may be made to copending applications U.S. Serial nos. 09/591,537 filed June 9, 2000 (atty docket TI-29884), 09/591,656 filed June 9, 2000 (atty docket TI-29960), and 09/932,794 filed August 17, 2001 (atty docket TI-31351). The stack (excluding the micro-stack 146), arrays and non-critical data may be stored in the D-cache 124, while Java local variables and associated pointers as explained below, as well as critical data and non-Java variables (e.g., C, C++) may be stored in D-RAMset 126. The instruction storage 130 may comprise instruction RAM ("I-RAMset") 132 and instruction cache ("I-cache") 134. The I-RAMset 132 may be used to store "complex" micro-sequenced Bytecodes or micro-sequences or predetermined sequences of code. The I-cache 134 may be used

to store other types of Java Bytecode and mixed Java/C-ISA instructions.

In accordance with a preferred embodiment of the invention, at least some applications executed by the JSM 102 comprise one or more methods. A "method" includes executable instructions and performs one or more functions. Other terms for "method" may include subroutines, code segments, and functions, and the term should not be used to narrow the scope of this disclosure.

A method (the "calling" method) may call another method (the "called" method). Once the called method performs its function, program control returns to the calling method. Multiple hierarchical levels of methods are possible as illustrated in Figure 5 which illustrates the interaction between three methods (Method A, Method B, and Method C). For purposes of the example of Figure 5, method A calls method B and method B calls method C. As such, method A is the calling method for method B which is the called method relative to method A and only while method A is executing. Similarly, method B is the calling method relative to method C which is considered the called method relative to method B.

A method may have one or more "local variables," as explained previously. Local variables may be used to temporarily store data or other information as the method performs its task(s). The local variables preferably are specific to the method to which the variables pertain. That is, method A's local variables ("LVA") are accessible generally by only method A and have meaning only to method A. Once method A completes, the method A local variables become meaningless. Similarly, LVB and LVC comprise local variables associated with methods B and C, respectively. Java Bytecodes refer to local variables using an index. The JVM maintains a local variables pointer ("PTR LV")

which points to the base address of the memory containing the current method's local variables. To access a particular local variable, a suitable index value is added to the base address to obtain the address of the desired local variable.

Figure 5 generally shows the state of the D-RAMset 126 in accordance with a time sequence of events 500, 510, and 520 as each method B and C is invoked. In sequence 500, method A is invoked and storage space 502 is allocated for its local variables (LVA). A base pointer (PTR LVA) 504 also is determined or selected to point to the base portion of LVA storage space 502. Using the pointer PTR LVA, references may be made to any local variable within method A's local variable set 502 by computing an index or offset to the PTR LVA value.

Although a plurality of methods may run on the JSM 102, typically only one method is "active" at a time having its instructions actively being executed by the JSM 102. The base pointer of the currently active method preferably is stored in register R5 as noted previously. In general, the base pointer for the active method may be computed by the JVM 108 while executing the invoke bytecode of the active method. This bytecode is a complex bytecode executed by a micro-sequence or by the JVM.

Sequence 510 depicts the state of the D-RAMset 126 when method A calls method B. In accordance with the preferred embodiments of the invention, the local variables (LVB) associated with method B are stacked in storage space 512 generally adjacent LVA ("on top of" LVA when viewed as in Figure 5). Following arrow 505, the base pointer for LVA (PTR LVA) preferably is also stored in the D-RAMset 126 adjacent (e.g., below) the LVB data at location 504A. Thus, the two local variable sets LVA and LVB may be separated by the base pointer (PTR LVA) for LVA and possibly other data. Once the base pointer 504 for LVA is stored adjacent

(below) the reserved space for the LVB data set 502, register R5 is updated (i.e., loaded) with a base pointer 514 for use with the LVB data set.

The JSM 102 may load LVA's base pointer 504 into location 504A by executing a store instruction to store LVA's base pointer at location 504A. Location 504A may be determined as the location pointed to by the base pointer of LVB (PTR LVB) minus 1. That is, the set of local variables associated with method B is mapped adjacent the pointer associated with method A's local variables. The value of PTR LVB may be determined as the sum of the value for PTR LVA 504, the size of LVA 502, and a value *p*. The value *p* may be an integer that is appropriate to take into account the size of the pointer itself and thus may be more than 4 bytes. Other data may be present between the storage areas for LVA 502 and LVB 512.

Following arrow 507 to time sequence 520, when method C is invoked (called by method B), the base pointer for method B (PTR LVB) is stored in location 514A which may be on top of LVB and below PTR LVC as shown and register R5 is loaded with the base pointer 524 (PTR LVC) to the base of the LVC data set 522. Method C's local variables (LVC) are allocated to storage space 522 which generally is adjacent (on top of) LVB 512 and PTR LVB 514A as shown. The PTR LVB value is stored in location 514A according to a similar calculation as that described above.

Figure 6 illustrates the return process as each method (Methods C and then B) completes and returns to its calling method (methods B and then A). Beginning with time sequence 530 in which the local variable frame comprises LVA, LVB, and LVC along with pointers PTR LVA and PTR LVB for LVA and LVB, method C completes. Control returns to method B and LVB's base pointer is loaded from location 514A into register R5 as shown by arrow 519 at time

sequence 532 by accessing PTR LVB through a load instruction that include a fixed offset from PTR LVC as a target address. Then, when method B completes, LVA's pointer (PTR LVA) is loaded into register R5 from location 504A as illustrated by arrow 521 at time sequence 534. The base pointers may be retrieved from their locations in cached-RAMset 126 by loading the value located at the location pointed by the currently active method's base pointer minus an offset (e.g., 1).

In accordance with preferred embodiments of the invention, the D-RAMset 126 is configured to provide any one or more or all of the following properties. The implementation of the D-RAMset 126 to provide these properties is explained in detail below. The local variables and pointers stored in the D-RAMset 126 preferably are "locked" in place meaning that, although the D-RAMset 126 is implemented as cache memory, eviction of the local variables generally can be prevented in a controlled manner. The locking nature of the D-RAMset 126 may be beneficial while a method executes to ensure that no cache miss penalty is incurred. Additionally, write back of valid, dirty local variables to main memory 106 is avoided in at least some situations (specified below). Further, mechanisms can be employed in the event that the D-RAMset 126 has insufficient capacity to accommodate all desired local variables. Further still, once a method has completed, the portion of the D-RAMset allocated for the completed method's local variables remains marked as "valid." In this way, if and when new methods are invoked and re-use the RAMset space (such as that described in one or more of the copending applications mentioned above, such methods' associated local variables will be mapped to the same portion of the D-RAMset. If the RAMset lines are already marked as valid, access to those new local variables does not generate any misses.

Retrieval of data from memory is unnecessary because the local variables only have significance while a method executes and a newly executing method first initializes all of its local variables before using them. Not generating misses and thus avoiding fetching lines from external memory reduces latency. After a relatively short period of time following the start of a Java program execution, all relevant lines of the RAMset are marked as valid and accesses to local variables of newly called methods do not generate misses, thereby providing superior performance of a "0-wait state memory." Furthermore, the cache properties of RAMset allow discarding or saving of the data in main memory whenever required.

In accordance with a preferred embodiment of the invention, the local variables (LVA-LVC) and associated pointers (PTR LVA-PTR LVC) may be stored in D-RAMset 126. The D-RAMset 126 may be implemented in accordance with the preferred embodiment described below and in U.S applications entitled "Cache with multiple fill modes," filed June 9, 2000, serial no. 09/591,656; "Smart cache," filed June 9, 2000, serial no. 09/591,537; and publication no. 2002/0065990.

As described in greater detail below, in the preferred embodiment, the data storage 122 (Figure 3) preferably comprises a 3-way cache with at least one cache way comprising D-RAMset 126. The D-RAMset (or simply "RAMset") cache 126 may be used to cache a contiguous block of memory (e.g., local variables and pointers as described above) starting from a main memory address location. The other two cache ways 124 may be configured as RAMset cache memories, or use another architecture as desired. For example, the data storage 122 may be configured as one RAMset cache 126 and a 2-way set associative cache 124. As such, the data storage 122 generally comprises one or more forms of cache

memory. The instruction storage 130 may be similarly configured if desired.

In operation, the processor's core 102 may access main memory 106 (Figure 1) within a given address space. If the information at a requested address in main memory 106 is also stored in the data storage 122, the data is retrieved from the data cache 124, 126. If the requested information is not stored in data cache, the data may be retrieved from the main memory 106 and the data cache 124, 126 may be updated with the retrieved data.

Figure 7 illustrates a more detailed block diagram of the data storage 122 in accordance with a preferred embodiment with a RAMset cache and a two-way set associative cache. A cache controller 222 may control operation of the data storage 122. Cache controller 222 may include a plurality of status bits including, without limitation, the following four status bits: RAM_fill_mode 224, Cache_Enable 226, DM/2SA 228 and Full_RAM_base 230 as well as other bits that are not specifically shown in Figure 7. The two-way associative cache may be configured as a direct map and its other way configured as a RAMset. Alternatively, the two-way set associative cache may be configured as two additional RAMsets depending on cache control bit DM/2SA 238 and FULL_RAM_Set_base 230 as described in at least one of the documents as referenced above. However, the preferred configuration comprises a single RAMset coupled to a standard data cache. The RAMset is not limited in size, nor must the RAMset have the same size as the other cache way. Therefore, if another RAMset is needed for capacity reasons, a single RAMset with a larger capacity may be preferred.

As shown, cache controller 222 couples to Full_Set_Tag registers 232 (individually referenced as registers 232a through 232c), Global_Valid bits 234 (individually referenced as bits 234a

through 234c), tag memories 236 (individually referenced as tag memories 236b and 236c), valid entry bit arrays 237 (individually referenced as bit arrays 237a through 237c) and data arrays 238 (individually referenced as data arrays 238a through 238c). Comparators 240 (individually referenced as comparators 240a through 240c) may couple to respective Full_Set_Tag registers 232. Comparators 242 (individually referenced as comparators 242b and 242c) couple to respective tag memories 236. Output buffers 244 (individually referenced as buffers 244a through 244c) may couple to respective data arrays 238. Hit/Miss logic 246 (individually referenced as logic 246a through 246c) may couple to comparators 240, global valid bits 234, valid bits 237, RAM_fill_mode bit 224 and Cache_Enable bit 226.

In operation, data storage 122 may be configured using the control bits 224, 226, 228 and 230. The Cache_Enable 226 allows the data storage to be enabled or disabled, as in standard cache architecture. If the data storage 122 is disabled (e.g., Cache Enable=0), data read accesses may be performed on the main memory 106 without using the data storage 122. If the data storage 122 is enabled (e.g., Cache_Enable=1), data may be accessed in the data storage 122, in cases where such data is present in the data storage. If a miss occurs, a line (e.g., 16 bytes) may be fetched from main memory 106 and provided to the core 120.

The size of the data array 238a may be different than the size of the data arrays 238 b, c for the other ways of the cache. For illustration purposes and without limiting this disclosure in any way, it will be assumed that data arrays 238b and 238c are each 8 Kbytes in size, configured as 512 lines, with each line holding eight two-byte data values. Data array 238a may be 16 Kbytes in size, configured as 1024 lines, each line holding eight, two byte data values. The ADDR[L] signals may be used to address one line

of the data array 238 and valid bit array 237 (and tag memory 236, where applicable). Accordingly, for the 1024-line first way, ADDR[L] may include 10 bits [13:4] of an address from the core. For the 512-line second and third ways, ADDR[L] may include 9 bits [12:4] of an address from the core. The ADDR[H] signals define which set is mapped to a line. Thus, assuming a 4 Gbyte address space, ADDR[H] uses bits [31:14] of an address from the core for the first way and uses bits [31:13] for each of the second and third ways of the cache 130.

The tag memories 236 and comparators 242 may be used for a 2-way set associative cache (e.g., D-cache 124 in Figure 3). When the core 120 performs a memory access, the tag memories 236 are accessed at the low order bits of the address (ADDR[L]). The tag memory locations store the high order address bits of the main memory address of the information stored in a corresponding line of the data array 238. These high order address bits may be compared with the high order address bits (ADDR[H]) of the address from the core 120. If the ADDR[H] matches the contents of the tag memory at ADDR[L], a hit occurs if the valid bit associated with the low order bits indicates that the cache entry is valid. If a cache hit occurs, the data from the corresponding data array 238 at ADDR[L] may be provided to the core 120 by enabling the corresponding output buffer 244. As described below, data from the 2-way cache is presented to the core 120 if there is a miss in the RAMset cache. By itself, the operation of the 2-way set associative cache and the direct map cache may be conventional and may not be affected by the RAMset cache 126. Other cache techniques could also be used in place of the 2-way cache 124.

The RAMset cache 126 preferably stores a contiguous block of main memory 106 starting at an address defined by the Full_set_tag

register 232 for the RAMset. This contiguous block of information (e.g., local variables/pointers) may be mapped to the corresponding data array 238 of the RAMset. In at least some embodiments, only the high order bits of the starting address are stored in the Full_set_tag register 232. Figure 8 illustrates this mapping for a single RAMset. As shown, the contents of Full_set_tag register 232a define the starting address for a contiguous block of memory cached in data array 238a.

Referring again to Figure 7, a RAMset miss may occur when the high order bits of the address from the core 120 do not match the contents of the Full_set_TAG register 232 or the global valid bit is "0". In either case, when a RAMset miss occurs, the data storage 122 may behave like conventional, 2-way cache logic. As such, if there is a hit in the 2-way associative cache, then data is presented to the core 120 from the 2-way set associative cache. Otherwise, the data is retrieved from main memory 106, forwarded to the core and loaded into a "victim" entry of the two-way associative cache.

A RAMset hit situation may occur when the high order bits of the address from the core 120 match the contents of the Full_set_TAG register 232 and the global valid bit equals "1" (the setting of the global valid bit is described in greater detail below). By default, the RAMset comparison preferably has higher priority than the other cache ways. A hit situation indicates that the requested data is mapped into the RAMset. If the Valid entry bit 237 corresponding to the line containing the data is set to "1", comparator 240 causes hit/miss logic 246 to generate a "hit-hit" signal because the address hit the RAMset and the data is present in the RAMset. If the corresponding valid bit 237 of the RAMset entry is "0", logic 240 generates a "hit-miss" because the address hit the RAM set, but the data is not yet present in the

RAM set. In this latter case, the data may be fetched from main memory 106 and loaded into the data array 238 of the RAMset. A hit in the RAMset logic preferably takes precedence over the normal cache logic. The standard logic of the 2-way cache generates a miss when the RAMset logic generates a hit. Information can reside in both the RAMset and the 2-way cache without causing any misbehavior; the duplicated cache entry in the 2-way cache will eventually be evicted by the replacement mechanism of the two-way cache because such data will not be used. When configured as a RAMset, data array 238 a, b, c can be configured as a local RAM or as a cached segment depending on the setting of a suitable configuration bit. However, even when configured as a local RAM, individual valid bits may be updated and misses do not generate accesses to the external memory.

To configure a RAMset for operation, the Full_set_tag register 232 preferably is loaded with a start address (set_start_addr) and the RAM_fill_mode bit 224 is configured to a desired fill mode. The circuitry for filling the cache can be the same as that used to fill lines of the set associative cache. At least one fill mode may be implemented and is referred to as a "line-by-line" fill mode as described below. Other fill modes may be implemented if desired such as the "set fill" mode described in at least one of the documents as referenced above.

For the line-by-line fill (RAM_fill_mode=0), the global valid bit 34 is set to "1" and each of the valid entry bits 237 is set to "0" when the Full_set_tag register 232 is loaded with the starting address. At this point, the data array 238 is empty (it is assumed that the Cache_Enable bit 226 is set to "1" to allow operation of the data storage 122). Upon receiving an address from the core 120, a valid entry bit 237 is selected based on the low order bits of the address. As provided above, if the RAMset

is 16 Kbytes in size, organized as an array of 1K x 16 bytes, where 16 bytes is equivalent to a block line in the associated 2-way cache, the Full_set_TAG register 232 may store 18 bits [31:14] of the starting address (set_start_addr). The address indexing each entry of the RAMset (ADDR[L]) may include 10 bits [13:4] while the data address used to access one data value in the line may include 3 bits [3:1] (assuming data accesses are 1 byte). A line of the data array 238 (at ADDR[L]) is loaded from main memory 106 each time that a hit-miss situation occurs because (1) the comparator 240 determines a match between ADDR[H] and Set_start_addr, (2) the Global valid bit 34 is set to "1" and (3) the valid bit 237 associated with the line at ADDR[L] is "0". This situation indicates that the selected line is mapped to the RAMset, but has not yet been loaded into the RAMset's data array 238. When the line is loaded into the data array 238 from main memory 106, the valid bit 237 corresponding to the line is set to "1".

This loading procedure (resulting in the valid bit being set to indicate the presence of valid data) has the same time penalty as a normal cache line load, but the entry will remain locked in the RAMset (i.e., the valid bit will remain set) and, therefore, the processing device will not be penalized on a subsequent access. As such, the lines used by a completed method remain valid so that re-using the lines by subsequent methods does not necessitate accesses to main memory 106. Further, freeing the local variable space for a completed method generally only involves disregarding the relevant base pointer. Further still, there is no need to copy back local variables upon to main memory 106 upon completion of a method because such extinct local variables are not used any more.

Upon completion of a method, the lines containing that method's local variables may remain marked as valid. As noted above, maintaining such lines marked as valid avoids generating misses in calls of new methods.

In accordance with at least some embodiments of the invention, a change of "context" may occur. In general, a context refers to the state of the processor that needs to be saved and eventually restored to be able to continue the execution of the same task at a later time. In Java and many other languages, a software process comprises independent software execution units, called "threads" that share the same memory space and have their own contexts. The local variables as well as the processor registers belong to the context. One or more methods and their associated local variables may be executed in one thread and the same or different methods may be executed in a different thread. Upon a change from a first thread to a second thread, the preferred embodiments of the invention cause local variables associated with the first thread and stored in the data array 238 that is configured as a RAMset to be stored off to external memory (e.g., memory 106 under the control of the JVM 108) for subsequent use if and when control changes from the second thread back to the first thread. However, in accordance with the preferred embodiments, not all local variables stored in a RAMset are necessarily copied to external memory upon a thread change. For example, local variables associated with unfinished (i.e., not yet completed) methods are copied to external memory, while local variables associated with finished (i.e., completed) methods are not copied to external memory. As explained previously, local variables cease to have meaning upon completion of the method that used the local variables. Thus, local variables associated with finished methods need not be copied to external memory upon

a thread change. The following discussion provides two embodiments of how to keep track and optimize the memory accesses of local variables associated with finished and unfinished methods.

Figure 9 illustrates one possible embodiment of how to differentiate between local variables associated with finished and unfinished methods. As shown, data array 128 generally includes two groups 300 and 302 of local variables. Local variable group 300 generally corresponds to methods that have not yet finished and thus still have significance. Local variable group 302 generally corresponds to methods that have finished and thus do not have any significance.

In the embodiment of Figure 9, the base address 304 comprises an address that points to the bottom of the data array 238 and is stored in the Full_set_tag register 232 which defines the contiguous block of memory that is mapped to the data array 238 as explained previously. A "top" address pointer 306 generally points to the boundary between the two local variable groups 300, 302. The top address 306 may be stored in any of the JSM's general purpose registers.

As methods are invoked in a thread, the invoke Bytecode (which may be implemented as a micro-sequence) sets the top address 306 to the top of the local variables that are allocated to the data array 238. As a method finishes, the top address 306 is adjusted to exclude the local variables associated with the now finished method. As such, the top address is effectively moved up and down through the data array 238 as methods are invoked and finished.

Upon a thread change (e.g., from a first thread to a second thread), a "clean range" Bytecode may be is executed to store the local variables residing on dirty lines within the given range

into the main memory and a "flush D_RAMset" Bytecode is executed to invalidate the entire data array for use by the second thread. The clean range Bytecode preferably clears the dirty bit in the given range (not specifically shown) associated with each entry in the data array 238. The flush D_RAMset bytecode clears the valid bits. Also, the top address 306 generally is saved in external memory for subsequent use. The second thread then may use the data array 238 for storing its own local variables. Upon changing back to the first thread, the clean range and flush D_RAMset Bytecodes again are executed to again clean the appropriate lines and to invalidate the entire data array. Further, the previously saved top address is retrieved from external memory and loaded into the JSM register dedicated for this purpose. Although as described above two status bits per line may be used, in other embodiments, a single status bit is used to distinguish used and unused lines.

Under control now of the first thread, methods may be executed on the JSM 102. Such methods may comprise one or more previously unfinished methods as well as new methods that are invoked following the change back to the first thread. Local variables associated with the first thread's previously unfinished methods should be retrieved from external memory via a "miss" technique. For example, upon a write miss to a line containing a targeted local variable in the unfinished group 300, the line containing the targeted local variable is fetched from external memory to the corresponding location in group 300 before the write is permitted to complete. The cache controller 222 preferably compares the targeted address to the top address 306 that has been reloaded into a suitable JSM register. If the targeted address is below the top address, then the cache controller 222 fetches the target value from external memory before completing

the write. On the other hand, if the cache controller determines that a write miss targets a line that is located above the top address (i.e., corresponding to a finished method), then the cache controller 222 precludes a fetch from external memory from occurring. Thus, fetches of local variables from external memory are permitted to occur depending on the targeted data array line relative to a threshold address (top) to minimize the number of unnecessary external memory accesses. Naturally, references to "below" and "above" the top address are implementation specific and thus may be altered (e.g., reversed) in accordance with other implementations. On the other hand, read misses may occur only for unfinished methods since local variables are written before being read. A read miss to an address below the top address generates a read access to the external memory and the corresponding line is placed in the D_RAMset and marked as valid.

Figure 10 illustrates an alternative embodiment of avoiding unnecessary external memory fetches upon a thread change. In Figure 10, instead of using an address to identify the boundary between the unfinished and finished local variable groups 300 and 302, an allocation bit 310 is included for each entry in the data array 238. A valid bit 237 is also shown associated with each entry as was discussed previously. In Figure 10, as was the case for Figure 9, local variable groups 300, 302 each may include one or more entries and each entry includes a valid bit 237 and an allocation bit 310. The valid bit 237 indicates whether the associated entry contains valid data. A valid bit set to a value of "1" may signify valid data, while a valid bit set to a value of "0" may signify invalid data. The allocation bit 310 preferably indicates whether a fetch from external memory is needed before completing the access to the associated line. An

allocation bit set to a value of "1" may signify that a fetch from external memory is needed, while a setting of "0" may indicate that a memory fetch is not needed. The use of the valid and allocation bits 237, 310 to determine when fetches from memory are needed is described below.

When data is stored in a line in the data array 238, the cache controller 222 preferably sets the associated valid bit 237 to "1." Further, when a method is invoked, the invoke process preferably sets the allocation bits 310 associated with the lines allocated to the newly invoked method to a value of "1." Then, when a method is finished, the allocation bits 310 associated with that method's local variables are set to a value of "0." Thus, in Figure 10, the local variables in groups 300 and 302 are all marked as valid. Further, the allocation bits 310 associated with the local variables in group 302 are set to a value of "0" thereby indicating that those local variables need not be fetched from memory upon a thread change. The allocation bits 310 associated with the group 300 local variables are set to a value of "1" to indicate that those local variables are to be fetched from memory upon a subsequent thread change.

The data array 238 depicts, for example, the state of the data array during execution of the first thread. As discussed above, control may change from the first thread to a second thread and back again to the first thread. The state of the allocation bits 310 are saved upon changing from one thread to another. Figure 11 illustrates the state of the data array 238 upon a change back to the first thread. The clean range command is performed to save dirty lines with the allocate bit set and to invalidate the entire data array 238 (i.e., set all valid bits 237 to 0). The allocation bits 310 were previously saved and thus are reloaded so as to be the same as shown in Figure 10.

In accordance with the embodiment of Figures 10 and 11, upon encountering an access (e.g., a write) to a line in groups 300 or 302, the cache controller 222 responds depending upon the state of the valid and allocation bits 237 and 310. The valid bit being 0 generally indicates the lack of valid data in the targeted line. The state of the allocation bit 310 dictates whether the targeted data value is fetched from external memory. If the allocation bit 310 is set to a value of 0 (local variable group 302), the cache controller 222 does not permit a fetch from external memory to occur. If, however, the allocation bit 310 is set to a value of 1 (local variable group 300), the cache controller 222 preferably causes a fetch from external memory to occur before permitting the write access to complete. In either case, once the data array 238 access completes, the associated valid bit 237 is set to a value of 1 to indicate the presence of valid data. Subsequent accesses to the same, now valid, line will not result in an external memory fetch because the targeted line is marked as containing valid data and thus no external memory fetch is needed. In this case, the state of the allocation bit may be ignored.

While the preferred embodiments of the present invention have been shown and described, modifications thereof can be made by one skilled in the art without departing from the spirit and teachings of the invention. The embodiments described herein are exemplary only, and are not intended to be limiting. Many variations and modifications of the invention disclosed herein are possible and are within the scope of the invention. Accordingly, the scope of protection is not limited by the description set out above. Each and every claim is incorporated into the specification as an embodiment of the present invention.

CLAIMS

What is claimed is:

1. A cache subsystem, comprising:

a cache controller; and

a data memory coupled to the cache controller, the data memory holds a contiguous block of memory defined by an address stored in a register accessible to the cache controller;

wherein the data memory is adapted to store two groups of local variables, a first group comprising local variables associated with finished methods and a second group comprising local variables associated with unfinished methods; and

wherein, based on a threshold value, local variables are fetched from memory that is external to the cache subsystem or saved to the external memory.

2. The cache subsystem of claim 1 wherein the threshold value comprises an address that points to a line in the data memory that separates the first and second groups.

3. The cache subsystem of claim 1 or claim 2 wherein the threshold value is determined from allocation bits associated with each of a plurality of lines comprising the data memory.

4. The cache subsystem of claim 2 or claim 3 wherein a reference to local variables is compared to said threshold address and a local variable from external memory is fetched if said reference is to an invalid line and is below said threshold address and a local variable is not fetched from external memory if said reference is above said threshold address.

5. The cache subsystem of claim 2, claim 3 or claim 4 wherein methods are invoked and each time a method is invoked, the threshold address is adjusted.

6. The cache subsystem of claim 5 wherein the threshold address also is adjusted when a method finishes.

7. The cache subsystem of any preceding claim wherein said data memory comprises a plurality of lines and said value comprises an allocation bit associated with each line.

8. The cache subsystem of claim 7 wherein each line also includes a valid bit and local variables are fetched from memory depending on the valid and allocation bits.

9. The cache subsystem of claim 7 or claim 8 wherein the allocation bits are set to one value for local variables associated with finished methods and to another value for local variables associated with unfinished methods.

10. A method, comprising:

allocating space in a data memory in which variables are stored that are used by methods;

setting a value indicative of which variables are used by finished methods and which variables are used by unfinished methods; and

wherein, based on the value, fetching local variables from external memory that is separate from the data memory or saving local variables to the external memory.

11. The method of claim 10 wherein setting the value includes adjusting a pointer address.

12. The method of claim 10 or claim 11 wherein setting the value includes setting a plurality of allocation bits associated with a plurality of lines provided in the data memory.

MEMORY MANAGEMENT OF LOCAL VARIABLES UPON A CHANGE OF CONTEXT

ABSTRACT OF THE DISCLOSURE

A cache subsystem may comprise a multi-way set associative cache and a data memory that holds a contiguous block of memory defined by an address stored in a register. Local variables (e.g., Java local variables) may be stored in the data memory. The data memory preferably is adapted to store two groups of local variables. A first group comprises local variables associated with finished methods and a second group comprises local variables associated with unfinished methods. Further, local variables are saved to, or fetched from, external memory upon a context change based on a threshold value differentiating the first and second groups. The first value may comprise a threshold address or an allocation bit associated with each of a plurality of lines forming the data memory.

Figure 10

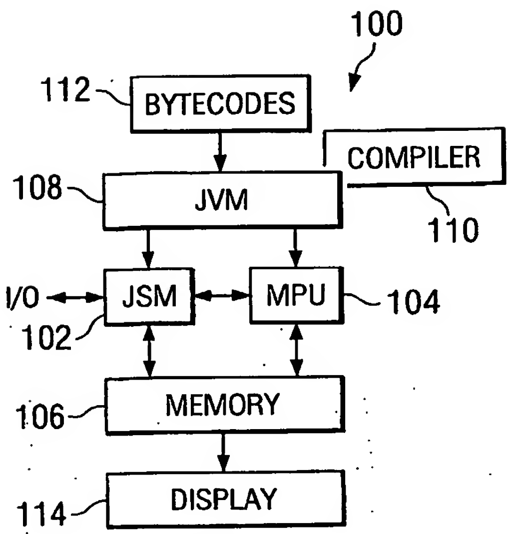

FIG. 1

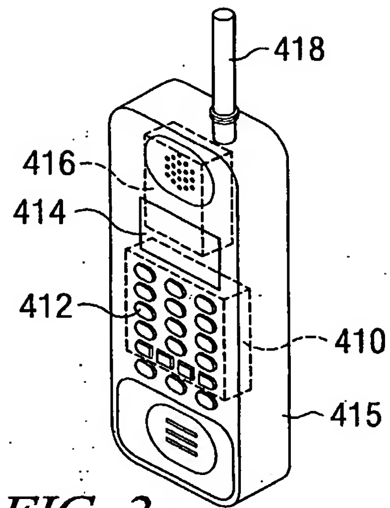

FIG. 2

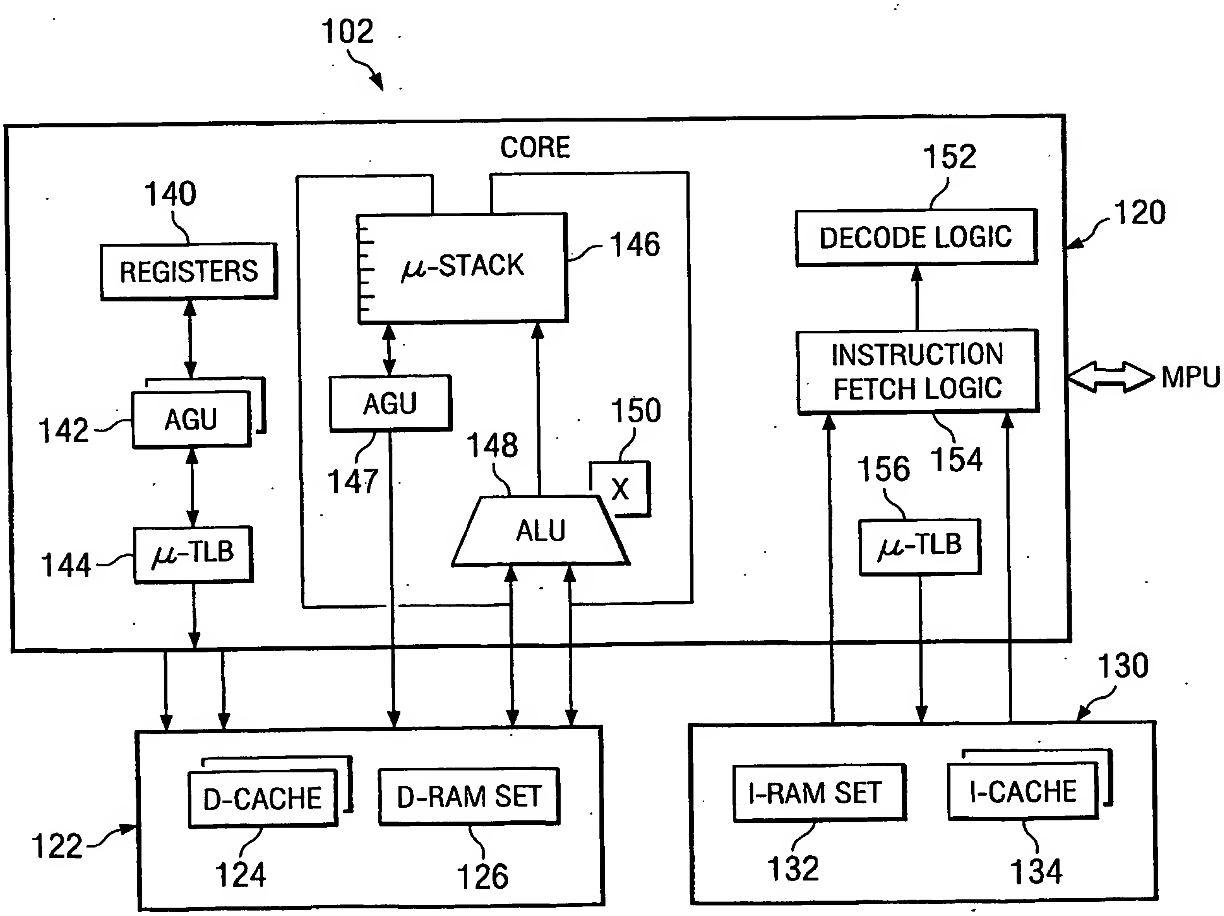

FIG. 3

R0	GENERAL PURPOSE (GP)
R1	GENERAL PURPOSE (GP)
R2	GENERAL PURPOSE (GP)
R3	GENERAL PURPOSE (GP)
R4	PROGRAM COUNTER (PC)
R5	GENERAL PURPOSE / LOCAL VARIABLE POINTER (LV)
R6	STACK POINTER (SP)
R7	TOP OF STACK (ToS)
R8	GENERAL PURPOSE / ADDRESS INDEX 0 (AI0)
R9	GENERAL PURPOSE / ADDRESS INDEX 1 (AI1)
R10	GENERAL PURPOSE (GP)
R11	GENERAL PURPOSE (GP)
R12	MICRO-PROGRAM COUNTER (MICRO-PC)
R13	GENERAL PURPOSE (GP)
R14	GENERAL PURPOSE / INDIRECT REGISTER INDEX (IRI)
R15	STATUS AND CONTROL (ST)

FIG. 4

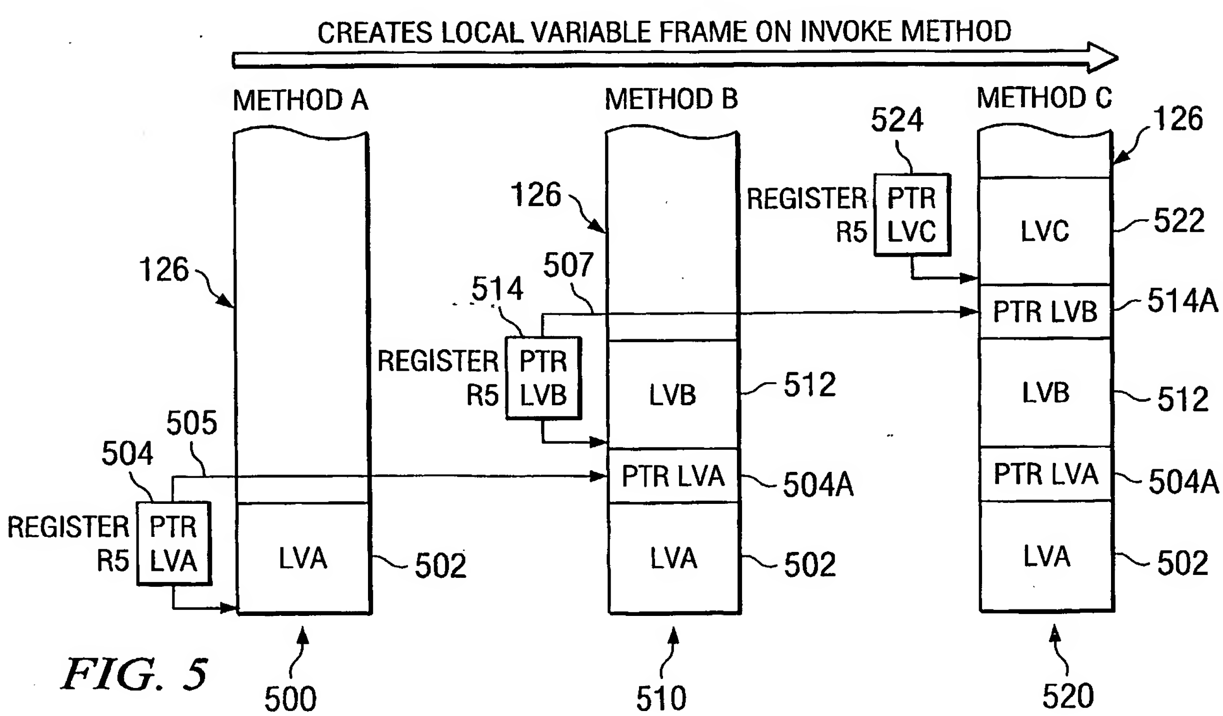

FIG. 5

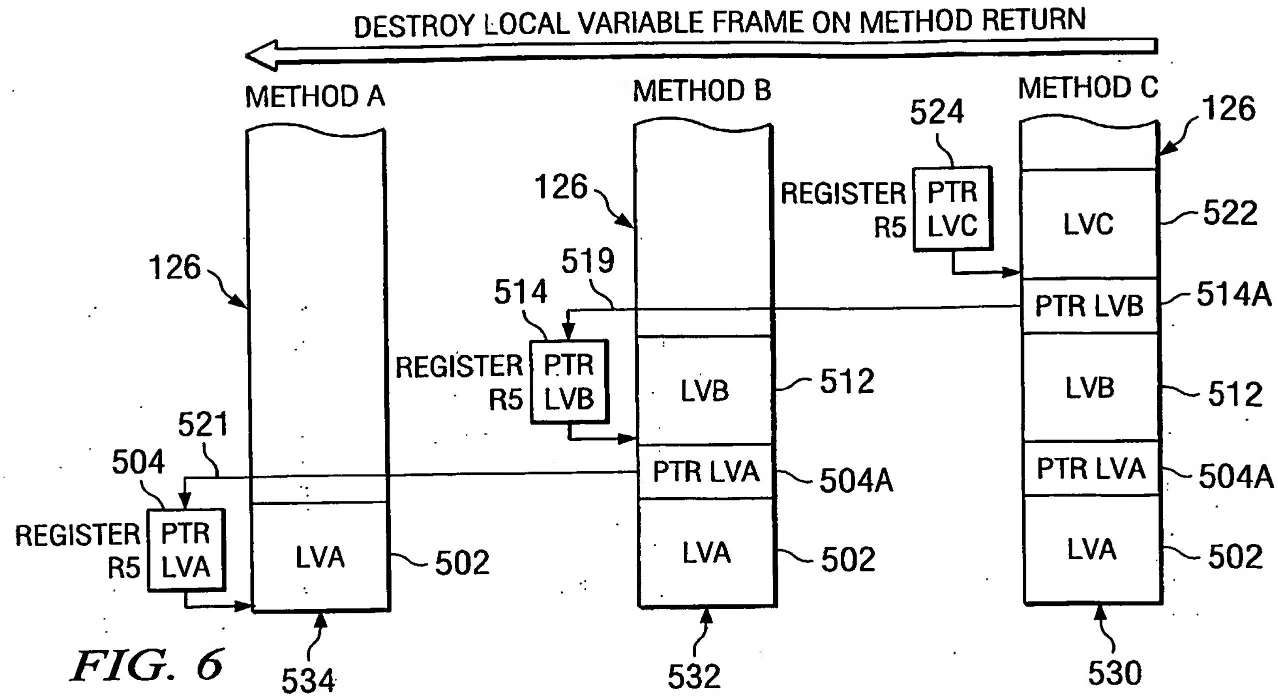

FIG. 6

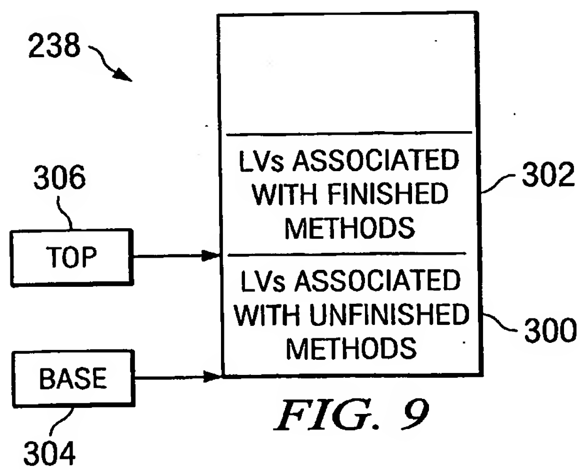

FIG. 9

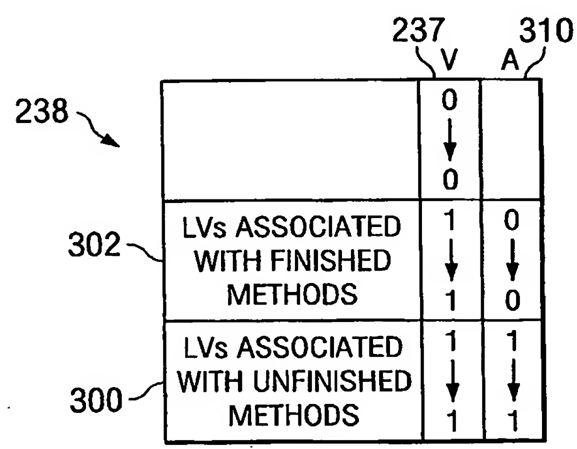

FIG. 10

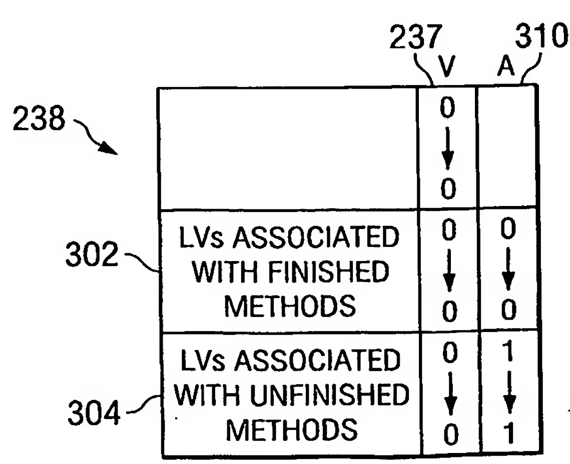

FIG. 11

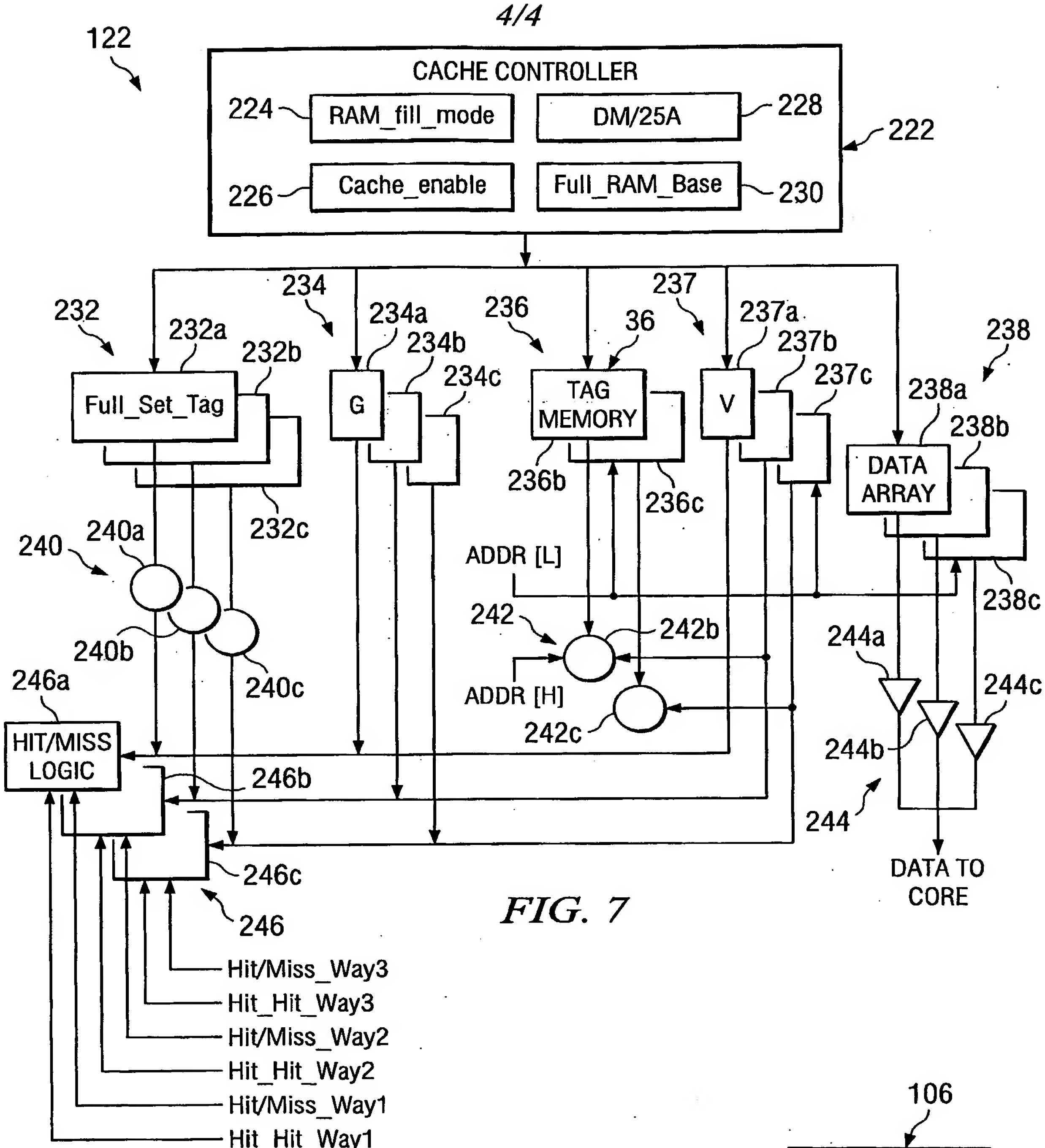
122
CACHE CONTROLLER
224
RAM_fill_mode
DM/25A
228
222
226
Cache_enable
Full_RAM_Base
230
232
232a
232b
234
234a
234b
234c
236
36
237
237a
237b
237c
238
238a
238b
Full_Set_Tag
G
TAG MEMORY
V
DATA ARRAY
236b
236c
232c
240
240a
ADDR [L]
238c
242
242b
244a
240b
240c
ADDR [H]
244c
246a
HIT/MISS LOGIC
242c
244b
246b
244
246c
DATA TO CORE
246
Hit/Miss_Way3
Hit_Hit_Way3
Hit/Miss_Way2
Hit_Hit_Way2
Hit/Miss_Way1
Hit_Hit_Way1

FIG. 7

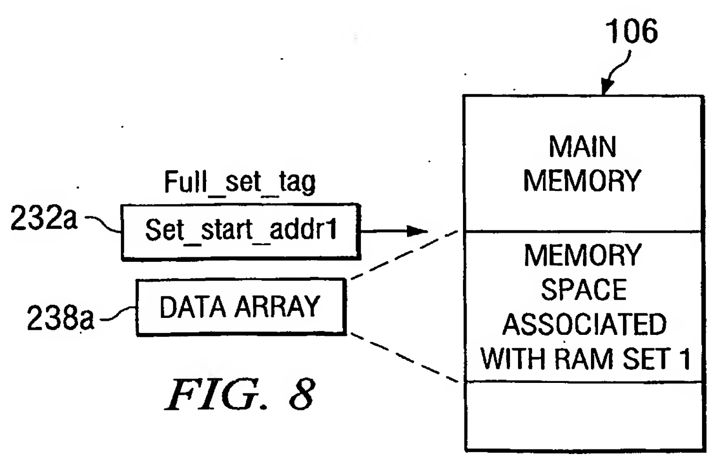
106
MAIN MEMORY
Full_set_tag
232a
Set_start_addr1
MEMORY SPACE ASSOCIATED WITH RAM SET 1
238a
DATA ARRAY

FIG. 8